Week 1 - Practical aspects of Deep Learning

07 June 2020 16:06

Setting up your Machine Learning Application.

TRAIN / DEV / TEST Sets

Layers
Hidden units
Activation fx
Learning rates ....

Predict good parameters.

data = [Train (60%) | dev / CV / Cross Validation Set (20%) | Test (20%)]

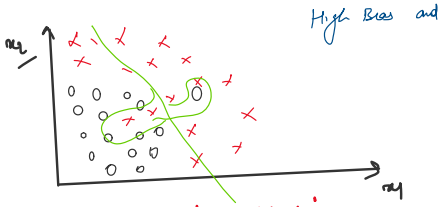
Not having a test set is okay and only have Dev Set;

BIAS / VARIANCE

Train set Error :→ 1%
Dev set Error :→ 11%

✓ Overfitting / High Variance
✗

High Bias and High Variance



Basic Recipe for Machine Learning

(1) High Bias ?? (Training data performance) ——Y——→
- Bigger N/w Layer
- Layer optimization Algos

(2) High Variance ? ——→ Y ——→ (Dev set performance)
- More data
- Regularization
- Neural N/w Architecture

Done!

" bias / Variance Trade off

bias ↑ Variance ↓

Regularizing your Neural N/w

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(w_{ij}^{[l]}\right)^2$$

Regularization

Logistic Regression —

$$\min_{(w,b)} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2}\|w\|^2$$

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

L2 Regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L1 Regularization: $\frac{\lambda}{m}\|w\|_1$ (λ = Regularization parameter)

L2 Regularization → $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L1 Regularization → $\frac{\lambda}{m} \sum_{i=1}^{n_x} |w| = \frac{\lambda}{m} \|w\|_1$

(λ = Regularization parameter)
[Hyperparameter]

Neural N/w

$J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|^2$

$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$ } "Frobenius norm" $\|\cdot\|_F^2$

$w \in (n^{[l]}, n^{[l-1]})$

$dw = (\text{Calculated from Backprop}) + \frac{\lambda}{m} w^{[l]}$

$\frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$

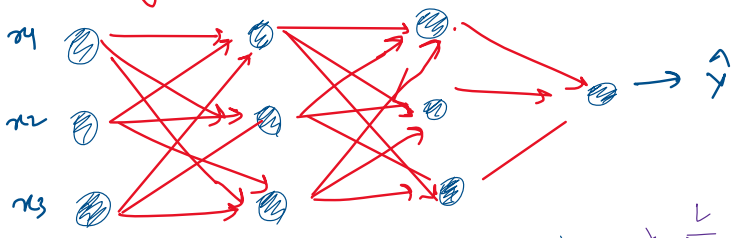
$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$

\# L2 regularization is also called "weight decay"

$w^{[l]} := w^{[l]} - \alpha \left[(\text{Backprop}) + \frac{\lambda}{m} w^{[l]} \right]$

$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha [(\text{Backprop})]$

Why Regularization prevents Overfitting ?

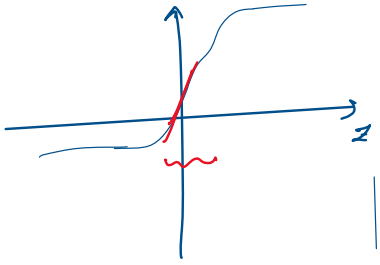


$J(w^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|W^{[l]}\|_F^2$

⊙ (If λ » big, $w \approx 0$ almost)

⊙ high bias → high variance

tanh(fx)



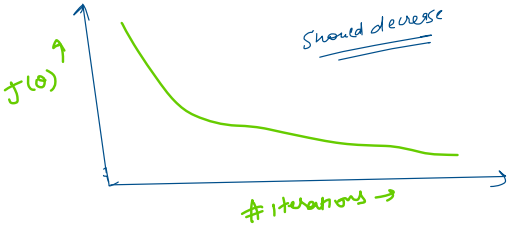
$g(z) = \tanh(z)$

$w^{[l]} \downarrow$, $\lambda \uparrow$

$f(x)$ will be linear in red range

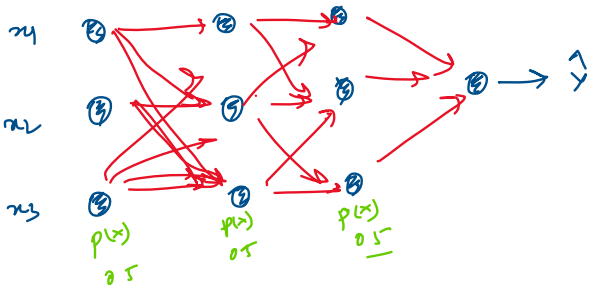
\# As if every layer is linear N/w

$J(\,) = \sum \mathcal{L}(\hat{y}, y) + \frac{\lambda}{2m} \sum_{l} \|W^{[l]}\|_F^2$



DROPOUT REGULARIZATION

(#) Set some probability to dropout a node in n/w,



Implementing Dropout (Inverted dropout)

illustrate in layer $l=3$:

$d_3 = np.random.rand(a_3.shape[0], a_3.shape[1]) < keep\_prob$

keep_prob = 0.8 (say)

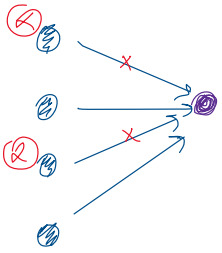
$a_3 = np.multiply(a_3, d_3)$ ($a_3 *= d_3$)

$a_3 /= keep\_prob$ (Inverted dropout technique)

(#) (don't dropout at test time.)

Understanding Dropout

(#) Can't rely on one feature, so have to spread weights
≈ Shrink weights



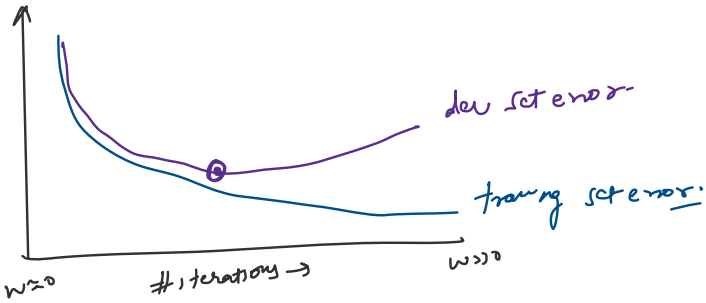
(i) Can vary keep prop according to layer,

units $\propto \frac{1}{keep\ prop\ value}$

(#) Computer vision

Other Regularization Methods

EARLY STOPPING



optimize cost fx J (Gradient descent, LBFGS, ...)
do not overfit
(Regularization, getting more data)

→ Orthogonalization → Thinking of one task at a time.

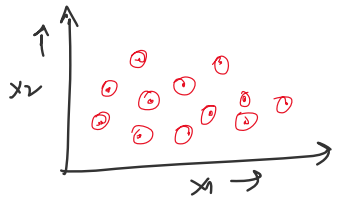
→ Orthogonalization → Thinking of one task at a time.

→ (-) No orthogonalization in early stopping

L_2 Regularization

Setting up your Neural N/W

Normalizing Inputs



$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

① Subtract Mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x := (x - \mu)$$

② Normalize Variances

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left[x^{(i)} \ast\ast 2\right] \quad \text{(Squaring)}$$

$$x = x/\sigma$$

$$\boxed{X = (x-\mu)/\sigma}$$

(-) use same μ and σ^2 to normalize Test set.

WHY NORMALIZE ?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

w_1 - 1 - 1000

w_2 - 1 - 10

Vanishing / Exploding gradients

$g(z) = z$

$\hat{y} = W^{[L]} * W^{[L-1]} * W^{[L-2]} * W^{[L-3]} * \cdots$

$W^{[3]}\ W^{[2]}\ W^{[1]}(X)$

$z^{(1)}$

$a^{(1)} = g(z^{(1)})$

$a^{(1)} = z^{1}$

$z^{(1)}$

$z^{[3]}$

$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ (1.5 | 0.5)

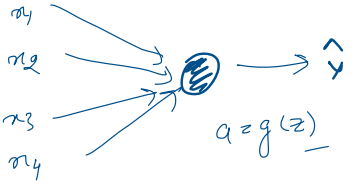
$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$

$\hat{y} = (1.5)^L$

* 1.5 times number of layers
* If you have deep neural n/w value of $\hat{y}$ will explode.

WEIGHT INITIALIZATION FOR NEURAL N/W.

Single Neuron



$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$

$b = 0$

$n \propto \frac{1}{w^i}$

$Var(w^i) = \frac{1}{n}$ [$n \in$ No of input features in neuron

$W^{[l]} = np.random.randint(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

$2/n$ for RELU (fx.)

$g^{(l)}(z) \in$ Relu

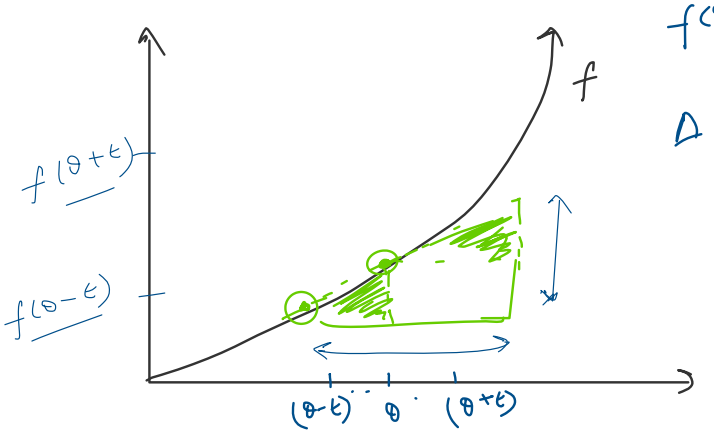
Other Variants

$\tanh(z) = \sqrt{1/n^{[l-1]}}$ { Xavier initialization. }

NUMERICAL APPROXIMATION OF GRADIENTS

Backprop (Gradient checking).

$f(\theta) = \theta^3$



$f(\theta) = \theta^3$

$$\Delta = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$$

$\approx g(\theta)$

GRADIENT CHECKING.

gradient checking for Neural N/W :→

Take $W^{[1]}, b^{[1]} \dots\dots W^{[L]} B^{[L]}$ & reshape to a big vector, θ.

(concatenate)

take $dW^{[1]}\ db^{[1]}, dW^{[2]}\ db^{[2]} \dots\dots dW^{[L]}\ db^{[L]}$ & reshape into a big vector, $d\theta$

(concatenate →)

grad check

for each i :→

$$d\theta_{approx}^{(i)} := \frac{J(\theta_1, \theta_2, \theta_3 \dots \theta_i + \epsilon) - J(\theta_1, \theta_2, \theta_3 \dots \theta_i - \epsilon)}{2\epsilon}$$

$\approx d\theta(i)$

$= \frac{\partial J}{\partial \theta_i}$

$$d\theta_{approx} \approx\approx d\theta$$

Check $= \frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \| + \| d\theta \|_2}$

$\epsilon \approx 10^{-7}$

Gradient Checking Implementation

(*) dont use grad check in training only to debug.

(*) if algorithm fails in grad check, look at the components & try to find the bug.

$db^{[l]}$, $dw^{[l]}$

(*) Remember Regularization

$$J(\theta) = \frac{1}{m} \sum \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l} \| w^{[l]} \|_2^2$$

$d\theta$ = grad of J wrt θ

(*) grad check doesn't work with dropout

(*) run at random initialization; perhaps after some training.

Week 2- Optimization Algorithms

08 June 2020 23:43

Mini batch gradient descent.

$X = [x^{(1)}, x^{(2)}, x^{(3)} \ldots x^{(m)}]$

(n_x, m)

$Y = [y^{(1)}, y^{(2)}, y^{(3)} \ldots y^{(m)}]$

$(1, m)$

if training examples are very large, say 10 million.

(#) will have to traverse (10 million) training set before you take one little step of gradient descent.

→ Mini-Batches

Now, One mini batch consists of = $\frac{\text{training-examples}}{\text{n. of mini-batches}} = n(mb)$.

if $X = 50$ (say) & $n = 5$

$X^{\{1\}} = [x^{(1)} \; x^{(2)} \; x^{(3)} \; x^{(4)} \ldots x^{(10)}]$

$X^{\{2\}} = \ldots$

→ and same for Y.

Mini-Batch t : $X^{\{t\}}, Y^{\{t\}}$

⊙ $X_{dims} = n_x, n(mb)$

⊙ $Y_{dims} = 1, n(mb)$

Mini-Batch gradient descent.

for $t = 1 \ldots n(mb)$:

// perform a step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$

(1) Perform forward prop. on

$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$\vdots$

$A^{[l]} = g^{[l]}(Z^{[l]})$

(2) Compute cost

$J = \frac{1}{m} \sum \mathcal{L}(\;) + \text{Regularization Term}$

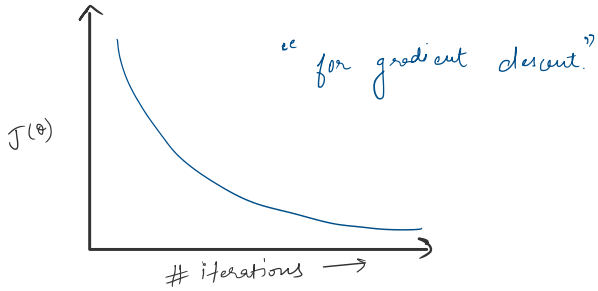
$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}$ & similarly,

$b^{[l]} := b^{[l]} - \alpha \, db^{[l]}$

" Also called 1 epoch / 1 pass

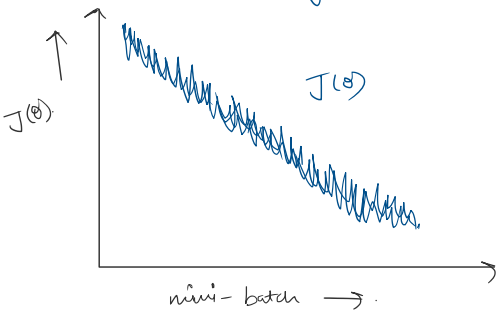
L2. UNDERSTANDING MINI BATCH GRADIENT DESCENT.

12. Understanding Mini Batch Gradient Descent.



"for gradient descent."

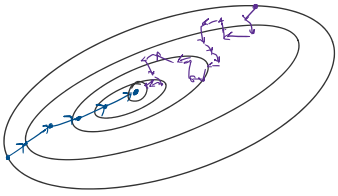
Mini – Batch gradient descent



Choosing size of mini batch !

* if mini batch size = m, then : Batch gradient descent.
* if (mini batch size = 1) : Stochastic gradient descent.

$(x^{\{1\}}, y^{\{1\}}, x^{\{2\}}, y^{\{2\}}, \ldots\ldots)$



[In practice mini batch size is between 1 and m.

Batch gradient descent	Stochastic gradient descent
mini - batch size : m	batch size : 1
Too long per iteration	loose speedup from vectorization.

→ mini - batch gradient descent

Choosing size of mini - batch

if Small training set : Use batch gradient descent.

if Small training set : Use batch gradient descent.
$(m \leq 2000)$

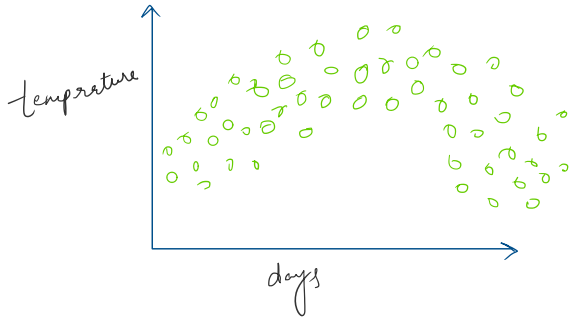
Typical mini-batch sizes : 2^n (multiple of 2.)

Make Sure mini-batch fits in CPU | GPU memory.
$X^{\{t\}}, Y^{\{t\}}$ → Hyperparameter.

13. EXPONENTIALLY WEIGHTED AVERAGE

Temperature in New-delhi :→

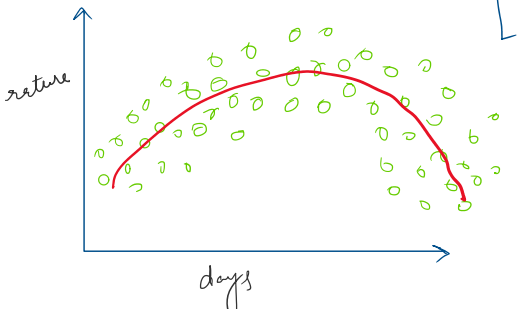
$\theta_1 = 32^\circ C$
$\theta_2 = 31^\circ C$
$\theta_3 = 29^\circ C$
$\theta_{180} = 28^\circ C$



$V_0 = 0$
$V_1 = 0.9 V_0 + 0.1 \theta_1$
$V_2 = 0.9 V_1 + 0.1 \theta_2$
$V_3 = 0.9 V_2 + 0.1 \theta_3$

$$V_t = 0.9 V_{(t-1)} + 0.1 \theta_t$$

→ Exponentially weighted average.
→ Moving Average.



$$V_t = \beta V_{(t-1)} + (1-\beta) C$$

Exponentially weighted moving average.

→ generalize formula.

$\beta = 0.9$

$V_t \overset{(avg)}{\approx} \frac{1}{(1-\beta)}$ days temperature.

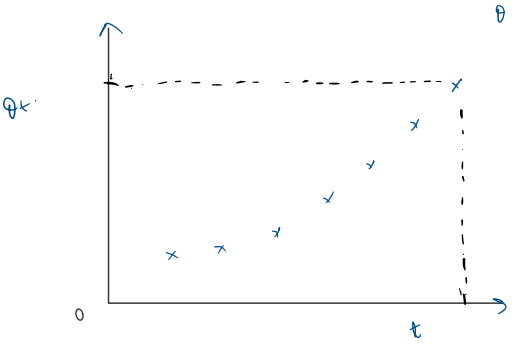
L4. Understanding Exponentially Weighted Avg.

$V_t = \beta V_{(t-1)} + (1-\beta)\theta_t$

$V_{100} = 0.1\theta_{100} + 0.9V_{99}$ $\quad \beta = 0.9$

$= 0.1\theta_{100} + 0.1 * 0.9(\theta_{99}) + 0.1(0.9)^2\theta_{98} + \ldots$

- exponentially decaying function.



(exp. decaying fx.)

$0.9^{10} \simeq 0.35 \simeq \frac{1}{e}$

$(1-\epsilon)^{1/\epsilon} = \frac{1}{\epsilon}$

Avg. over 10 days.

- $V_0 = 0$
- $V_t = \beta V_{(t-1)} + (1-\beta)\theta_1$

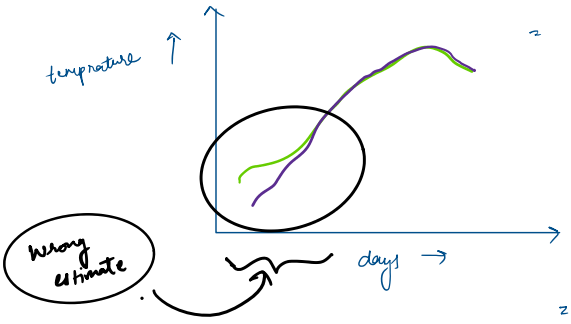
$V = 0$

$V_0 = \beta V + (1-\beta)\theta_1$

$V_0 = \beta V + (1-\beta)\theta_2$

L5. Bias Correction in Exp. Weighted Avg.

$V_t = \beta V_{t-1} + (1-\beta)\theta_t$; $\boxed{\beta = 0.98}$



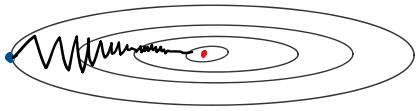
$= \frac{V_t}{(1-\beta^t)}$

$\boxed{t=2}$:

$= \frac{V_2}{1-\beta^2} = \frac{}{(1-0.98)^2}$

$= \frac{V_2}{0.0396}$

L6. Gradient Descent with Momentum



↕ — Slower learning.

↔ — faster learning.

Momentum :-

On every iteration t, Calculate :-

dw, db on current batch → friction

$V_{dw} = \beta V_{dw} + (1-\beta)dw$ ← Acceleration

weights

dw, db on current batch → fraction

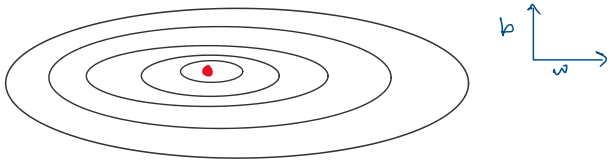
Velocity ← $V_{dw} = \beta V_{dw} + (1-\beta) dw$ } — Acceleration
$V_{db} = \beta V_{db} + (1-\beta) db$

$w := w - \alpha V_{dw}$

$b := b - \alpha V_{db}$

Hyperparameters :- α, β $\quad \boxed{\beta = 0.9}$

L7. RMS PROP (Root mean square prop)



On every iteration, t :-

Compute dw, db for mini-batch.

$S_{dw} = \beta S_{dw} + (1-\beta_2) dw^2$ — element wise square.

β is hyperparam for momentum

$S_{db} = \beta S_{db} + (1-\beta_2) db^2$

$S_{dw} \approx 0$

to make sure it's +ve, add a small value ϵ.

$\epsilon \approx 10^{-8}$

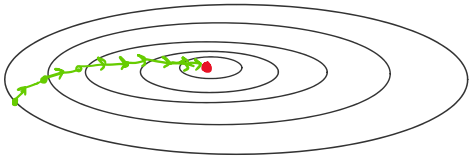
$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \epsilon}$

$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$

(*) $w \rightarrow$: fast.
$b \uparrow$: slow down.

S_{dw} = relatively small.

S_{db} = relatively large.



* Rms prop.

→ now, can also use large learning rate, α.

L8. ADAM's OPTMIZATION ALGORITHM

{ gd (Momentum) + rmsprop. }

$V_{dw} = 0$, $S_{dw} = 0$; $V_{db} = 0$ $S_{db} = 0$

On every iteration t :

* Compute dw, db using current mini-batch.
* $V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$
  $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$
  } • Momentum (Hp: β_1)
* $S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$
  $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$
  } • rms prop. (Hp: β_2)

Bias correction

* $V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$
  $V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$ } • Momentum

$S_{dw}^{corrected} = V_{\partial b} / (1-\beta_1^t)$
$S_{db}^{corrected} = V_{\partial b} / (1-\beta_1^t)$ } • Rms prop.

* $w := w - \alpha \left[\frac{V_{dw}^{corrected}}{\left(\sqrt{S_{dw}^{corrected}} + \epsilon\right)} \right]$

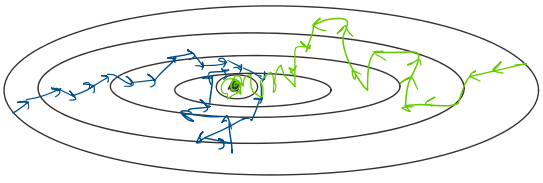
* $b := b - \alpha \left[\frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon} \right]$

Hyperparameters choice :-

α = needs to be tuned.
β_1 = 0.9 (dw)
β_2 = 0.999 (∂w^2, ∂b^2)
ϵ = 10^{-8}

ADAM : "Adaptive moment Estimation."

L9. LEARNING RATE DECAY :



Slowly reduce, α

* 1 epoch = 1 pass over the data.

$\alpha = \left(\frac{1}{1 + \text{decay rate} * \text{epoch num}} \right) \alpha_0$ → some initial learning rate.

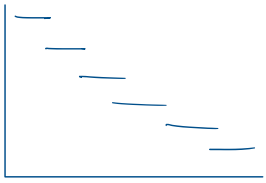
Epoch	α
1	0.1
2	0.067
3	0.5
4	0.4

$\alpha_0 = 0.2$
decay = 1

$\alpha = \left[(N<1)^{\text{epochnum.}} \cdot \alpha_0 \right]$:- exponential decay.

people also use:

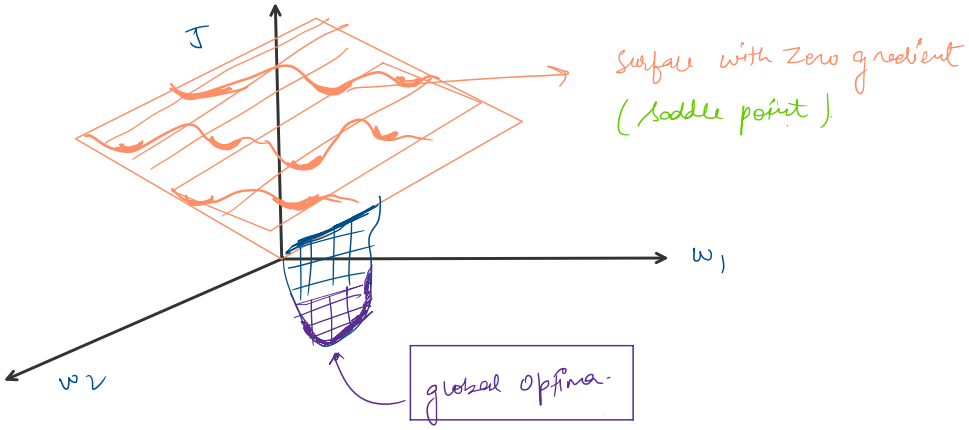
$\alpha = \frac{k}{\sqrt{\text{epochn.}}} * \alpha_0$



:- descrete staircase.

* Manual decay.

L10: The Problem Of Local Optima



Problem with plateaus.

* [plateau is a region where the derivative is 0 for a very long time.

- Unlikely to stuck in a bad local optima.
- Plateaus can make learning slower.

Week 3 - Hyperparameter tuning, Batch Normalization and programming frameworks

11 June 2020 17:26

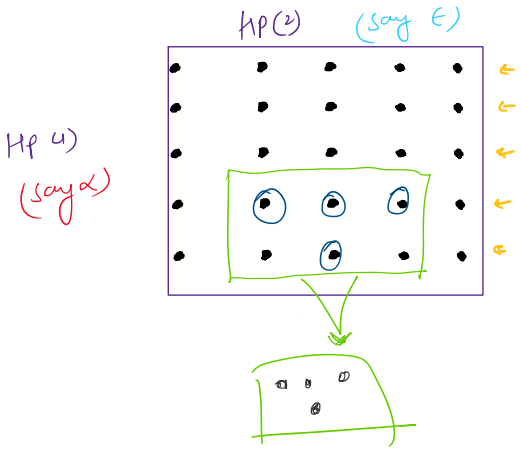
Hyperparameter Tuning

L1. Tuning process.

Hyperparameters.

α, β (~.9), β_1 (.9), β_2 (.999), ϵ (10^{-8}), #layers, #hidden units, learning rate decay, mini-batch size.

* - high priority.
* — Mediocre priority.
* — Low priority.
* — almost negligible (default values are used.)



25 values

{ Randomly pick these hp's and find that works best.

- Coarse to find sampling scheme.

* Use random Sampling and adequate search.
{optional} * Coarse to find search process.

L2. Using An Appropriate Scale To Pick HP's.

α 0.0001 ... 1

90% Values $\in [0.1, 1]$

using log scale

.0001 .001 .01 1

$a = \log_{10}(.0001) = -4$ (10^{-4})
$a = -4$
$r = -4 * np.random.rand()$

(10^0) $b = \log_{10}(1)$
$b = 1$
$r = [-4, 0]$

$a = \log_{10}(.001)$

$a = -4$

$r = -4 * np.random.rand()$

$\alpha = 10^r$

Sample r uniformly on $\in [a, b]$

$r = [-4, 0]$

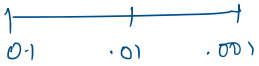
$\alpha = [10^{-4}, 10^{0}]$

Hp's for exponentially weighted avg.

$\beta = [0.9 \text{ ---- } 0.999]$

Linear Scale — ⊗

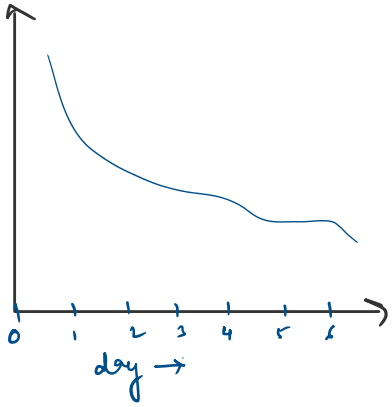
$1-\beta = [0.1 \text{ ---- } 0.001]$ — ✓



Very sensitive to small changes in β.

L3: HP tuning (Panda v/s Caviar)

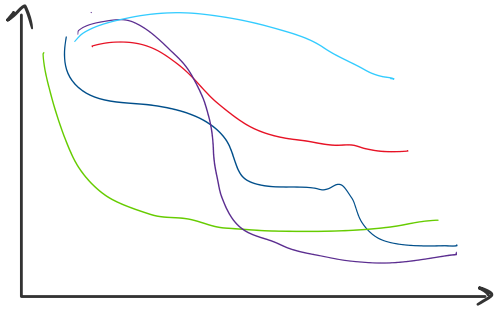
Babysitting one model.



Keep checking one day at a time.

Panda Approach.

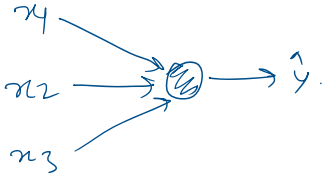
Training many models in Parallel



Caviar Approach.

Batch Normalization

L1: Normalizing Activations in a N/W



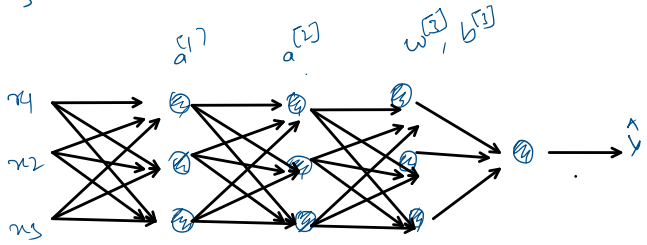
$X = X - \mu$

$\sigma^2 = \frac{1}{m} \sum_i x^{i\,2}$

$X /= \sigma^2$

$z^{[1]}$ $a^{[1]}$ $w^{[1]}, b^{[1]}$

x_3



Can we normalize $a^{[2]}$? to train $w^{[3]}$ $b^{[3]}$ faster ?

* Normalize $z^{[2]}$ or $z^{[2]}$ instead of $a^{[1]}$

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{normalize} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

z has normal distribution and Variance 1, but we want different dist.

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

Learnable parameter.

Standardize mean and Variance.

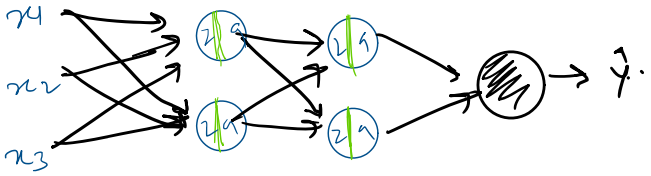
if $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$ {

then :

$$\tilde{z}^{(i)} = z^{(i)}$$

Use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$

L2. Fitting Batch Norm to a N/W :>



$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[(B \cdot N)]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g(\tilde{z}^{[1]}) \rightarrow z^{[2]} \cdots\cdots$$

Parameters :> $w^{[1]}, b^{[1]}, \ldots, \beta^{[1]}, \gamma^{[1]}$

$$\beta^{[1]} := \beta^{[1]} - \alpha \cdot d\beta^{[1]}$$

- **Working with mini-batches.**

$$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} \tilde{z}^{[1]} \xrightarrow[(B \cdot N)]{\beta, \gamma} g^{[1]}(\tilde{z}^{[1]}) \rightarrow \cdots\cdots$$

$g^{[1]}(\tilde{z}^{[1]}) = a$

$w^{[l]}, b^{[l]}$ → $(B \cdot N)$ → a

Parameters = $w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$ ⇒ $z^{[l]} = w^{[l]} * a^{[l-1]} + b^{[l]}$ (× ×)

* Batch Norm zeroes out all $z^{[l]}$, so adding/subtracting any any const. doesn't effect the equation.

$$z^{[l]} := w^{[l]} a^{[l-1]}$$

$$z^{[l]}_{norm} := \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]} + \beta^{[l]}$$

dims :- $(n^{[l]}, 1)$

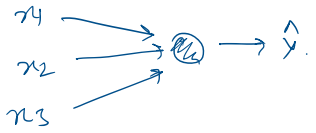
Implementing gradient descent

for t=1 to ... Mini-Batch(n) :-

- Compute forward prop on $X^{\{t\}}$
- use batch Norm to replace $z^{[l]} \rightleftharpoons \tilde{z}^{[l]}$
- use backprop to calculate grads.
- update parameters, → $w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$
  → $\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$
  → $\gamma^{[l]} := \gamma^{[l]} - \alpha\, d\gamma^{[l]}$

L3. Why does batch norm works?

* learning on shifting input dist.



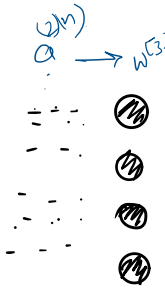
Trained on monochrome image. Trained on coloured image

— "Covariate Shift" →

* If you learn some $x \to y$ mapping, if distribution of x changes then you need to retrain your learning algorithm.

→ Covariate Shift in N.N

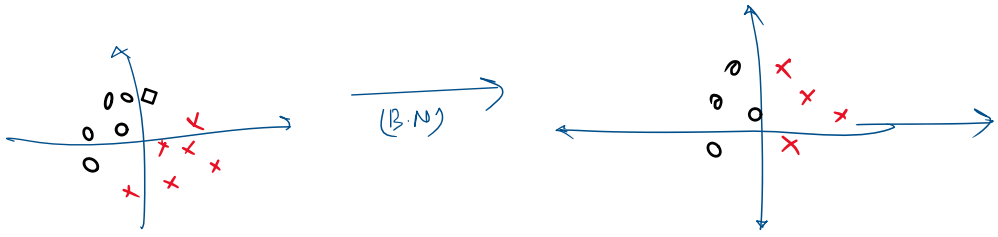
→ Covariate Shift in N.N



Problem

→ parameters w, b before this layer change, $a^{[2]}$ will also change. Suffering from a problem of "Covariate Shift".

"Batch norm : No matter the distribution changes, mean and variance remains same.



Batch Norm as regularization.

- Each mini-batch is scaled by mean/variance computed on just one batch.
- This adds some noise $z^{(i)}$, scaling process of $z^{[i]} \to \tilde{z}^{[i]}$ becomes more noisier, similar to dropout.
- This has slight regularization effect.

mini-batch size $\propto \frac{1}{\text{noise}} \propto$ regularization.

(*) "don't use batch norm as regularization only use for normalization." ✗ ✓

L4. Batch Norm at test time.

We need different μ and σ^2.

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

[m :- ex in mini-batch.

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$\to z^{(i)} = z^{(i)} - \mu$$

$$Z_{norm}^{(i)} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{Z}^{(i)} = \gamma Z_{norm} + \beta$$

estimate using exponentially weighted avg. (across mini-batch)

$X^{\{1\}} \to \mu^{\{1\}[1]}$ (θ_1) $\sigma^{2\{1\}[1]}$

$X^{[2]} \to \mu^{\{2\}[1]}$ (θ_2) $\sigma^{2\{2\}[1]}$

$X^{[3]} \to \mu^{\{3\}[1]}$ (θ_3) $\sigma^{2\{3\}[1]}$

---

new Z_{norm}

$$Z_{norm} = \frac{Z - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad \tilde{Z} = \gamma Z_{norm} + \beta$$

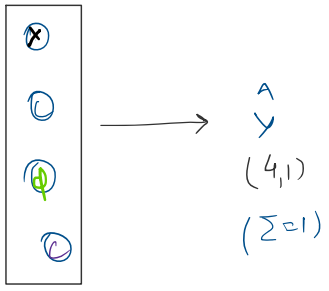
Multi Class Classification

4. Softmax Regression.

Now recognize cats, dogs and chicks.

Class1 Class2 Class3 -1.

classes : 4



$\hat{y}$

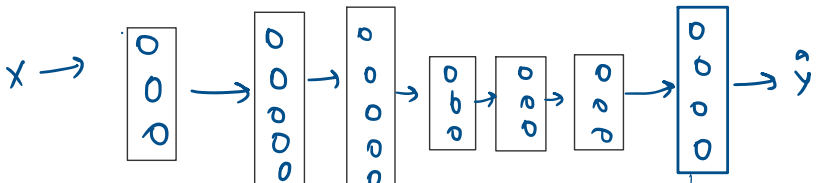
$(4,1)$

$(\Sigma = 1)$

— final layer —

Softmax Layer

Softmax Layer



$$z^{[L]} = W^{[L]} * a^{[L-1]} + b^{[L]}$$

Softmax Activation fx

$$t = e^{(z^{[L]})}$$

$(4,1)$

$$a^{[L]} = \frac{e^{(z^{[L]})}}{\sum_{i=1}^{4} t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

$(4,1)$

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20 \end{bmatrix}$$

$$\sum_{i=1}^{4} t_i = 176.3$$

$$a^{[L]} = \frac{t}{176.3} = \frac{1}{176.3} * \begin{bmatrix} e^5 \\ e^2 \\ e^1 \\ e^3 \end{bmatrix}$$

$$a^{[L]} = g^{[L]}(z^{[L]})$$

(4×1)

L2: Training a Softmax Classifier.

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5 / \sum t_i \\ e^2 / \sum t_i \\ e^1 / \sum t_i \\ e^3 / \sum t_i \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

"hard max"

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \} 0 \rightarrow \text{class with highest probability.}$$

* Softmax regression generalizes Logistic Regression to C classes.
  If $C = 2$, Softmax Regression = Logistic Regression.

Loss fx.

$y_{(4\times1)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ → Cat. (ground truth.)

$\hat{y}_{(4\times1)} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j (\log \hat{y}_j)$$

$(y_1 = y_3 = y_4 = 0)$: 3 values in $\mathcal{L}(\hat{y}, y) = 0$

$$\mathcal{L}(\hat{y}, y) = -y_2 \log \hat{y}_2$$

(minimize)

$y_2 = 1 \Rightarrow -\log \hat{y}_2$

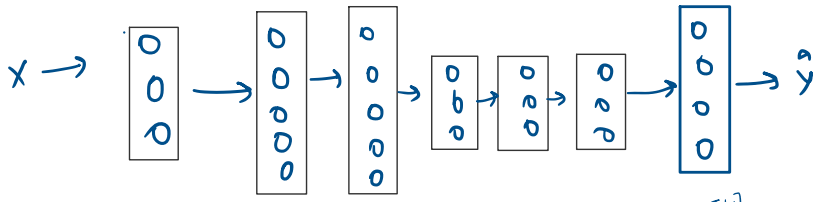
make $\hat{y}$ as big

$$J(w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$Y = [y^{(1)} \; y^{(2)} \; y^{(3)} \cdots y^{(m)}]_{(4\times m)}$$

$$\hat{Y} = [\hat{y}^{(1)}, \hat{y}^{(2)}, \hat{y}^{(3)} \cdots \hat{y}^{(m)}]_{(4\times m)}$$

Gradient Descent with Softmax.



$z^{[l]} \rightarrow a^{[l]} \rightarrow \mathcal{L}(-)$.

$(\hat{y})$

forward prop. : ✓

backward prop. :-

$dz^{[l]} = \hat{y} - y$. $(\in 4,1)$ $(C,1)$

$\frac{dJ}{dz^{[l]}}$

Deep Learning Frameworks

- Caffe / Caffe 2
- CNTK
- DL4J
- Keras.
- Lasagne
- mxnet
- Paddle Paddle
- → Tensorflow.
- Theano
- Pytorch.